Fast Library for Number Theory: an introduction

William B. Hart

Mathematics Institute, Warwick University, Coventry, United Kingdom

Abstract. We discuss FLINT (Fast Library for Number Theory), a library to support computations in number theory, including highly optimised routines for polynomial arithmetic and linear algebra in exact rings.

1 Introduction

The Fast Library for Number Theory (FLINT) [6] is a software library, written in highly optimised C, to support computations in number theory. Its initial scope is to cover the polynomial arithmetic and linear algebra functionality of a library like Victor Shoup's Number Theory Library (NTL), [16]. However, the eventual aim of FLINT will be to provide an alternative to the Pari library [2], with a focus on higher level computations in Algebraic Number Theory.

The design motivations for FLINT are that it be:

- Written entirely in C (some assembly optimisations)
- Threadsafe design
- Implement asymptotically fast algorithms where available
- As fast or faster than other Open Source and Proprietary options
- Completely Open Source (GPL licensed)

FLINT is constructed as a set of modules, each based around a given type, e.g. the fmpz_poly module, which is based around a type for polynomials with multiple precision integer coefficients (the FLINT fmpz type).

2 Basic Integer Arithmetic

FLINT supports integer arithmetic in three ways. Firstly, the fast polynomial multiplication code (see the next section) is used to provide very fast integer multiplication for operands above about 2000 limbs. This implementation is often faster than GMP [5] (which is used for smaller multiplications), by as much as 30%.

Secondly, FLINT offers a highly optimised multiple polynomial quadratic sieve, for factoring integers. This is efficient up to about 70 decimal digits and still much faster than Pari for larger factorisations.

Thirdly, FLINT's ulong_extras module provides fast code for operations involving C integers, i.e. long int's. This includes modular arithmetic, gcd, primality testing, efficient factorisation, via numerous optimised factoring routines. One innovation here is the One Line Factor algorithm, which is competitive with SQUFOF (also implemented). See [10] for details.

3 Polynomial Arithmetic

The bulk of code in FLINT supports polynomial arithmetic for multiprecision integer coefficients and for coefficients in $\mathbb{Z}/n\mathbb{Z}$ for up to machine word sized moduli.

FLINT implements numerous integer polynomial multiplication routines, including the classical and Karatsuba routines, Kronecker Segmentation and the Schoënhage-Strassen algorithm. The latter is based on highly optimised Fast Fourier Transform code, the final version of which was developed by David Harvey, based on the ideas presented in his paper [12].

Division of polynomials is achieved using a modified version of Mulders' algorithm (see [14], [11]), which is competitive with the usual middle product approach, but simpler to implement.

The polynomial modules also offer routines for power series operations, GCD, resultant, evaluation and composition. The latter is achieved with an algorithm implemented by Andy Novocin and the author [7].

The $\mathbb{Z}/n\mathbb{Z}$ module in the FLINT 1 series makes use of David Harvey's zn_poly library. This uses his fast Kronecker Segmentation variations [13] to achieve up to a 40% improvement over standard Kronecker Segmentation, and a highly optimised Schoënhage-Nussbaumer FFT implementation.

The $\mathbb{Z}/n\mathbb{Z}[x]$ module offers factorisation based on the Berlekamp and Cantor-Zassenhaus algorithms. The $\mathbb{Z}[x]$ module has a first implementation of the new factorisation algorithm of Novocin and van Hoeij [8] which improves on the original algorithms of van Hoeij, Belabas and others.

4 Linear Algebra

The linear algebra component of FLINT is still relatively young, providing some basic types and a FLINT rewrite of Damien Stehlé's fpLLL [15]. It also offers Hermite Normal Form and basic operations including matrix multiplication.

5 FLINT 2

The FLINT 2 series is a complete rewrite of FLINT 1 from scratch. Its focus is on very clean code and even better performance than FLINT 1. It also offers modules for multivariate polynomial arithmetic, polynomials over $\mathbb{Z}/n\mathbb{Z}$ for multiprecision n and optimised linear algebra over $\mathbb{Z}/nZ$ for word sized moduli. It should be released by the end of 2010.

6 Comparison with other libraries

The performance of FLINT, especially in polynomial arithmetic is usually equivalent to that of Magma [3] and in many cases faster (polynomial factoring over $\mathbb{Z}/nZ$ is still an exception to this). Magma is already up to five times faster than NTL even for polynomial multiplication.

The Pari library is not usually asymptotically fast and NTL is not threadsafe.

FLINT has been used to perform some very large computations, e.g. the computation of 10^{12} coefficients of the congruent number theta series [9].

7 Choice of language – C

In preparing this abstract the referee requested that the author comment on the suitability of C as a language for FLINT as compared with a higher level language, say. Indeed, C is a terrible choice because it is not low level enough (it does not support carry handling or return of the high word of a product - though recent versions of GCC allow the latter via extensions), it does not comfortably support generic programming and its syntax is complex and not well suited to mathematics.

However, C is also the best choice available because it is the clear frontrunner performance-wise. C is compiled, statically typed and the GNU compiler collection and some commercial C compilers have extremely sophisticated back ends, yielding good performance.

C++ would add Object Oriented programming, however its syntax is so complex that its front end actually rivals the back end in complexity. But most importantly, the C/C++ syntax cannot be extended, its macro processing being essentially search-and-replace only.

Lisp is a high level language which is formally syntactically extensible through macros, but uses uncomfortable prefix notation. Forth is a low level language which is formally syntactically extensible, but has equally uncomfortable postfix notation and requires the programmer to think in terms of a stack.

OCaml is properly syntactically extensible through its powerful preparser, but its native syntax is uncomfortable, and once again performance has fallen behind that of good C compilers.

The ideal language would have a syntactically extensible front end sitting on an already highly optimised back end, such as that of GCC's GENERIC or LLVM's IR. To our knowledge, such a language simply doesn't exist. Even with that sophistication, no existing back ends properly support carries, meaning that for some things, assembly language is still required.

We mention one other solution of note. Sage [4] is written in Python, which has a very clean syntax, but is very high level with terrible performance (often a hundred times slower than highly optimised C). For performance critical code, Sage makes use of Cython [1], a dialect of Python, which allows static C type declarations. These can be compiled to C, which for many situations will give native C performance. In fact, FLINT is used in Sage, and Cython is used extensively in the FLINT wrapper to ensure maximum efficiency.

References

1. Stefan Behnel, Robert Bradshaw, Dag Seljebotn, *Cython: C extensions for Python*, http://www.cython.org/.
2. Karim Belabas, *Pari/GP*, http://pari.math.u-bordeaux.fr/.
3. John Cannon, Allan Steel, et al., *Magma Computational Algebra System*, http://magma.maths.usyd.edu.au/magma/.
4. Burcin Erocal, William Stein, *The Sage Project: Unifying Free Mathematical Software to Create a Viable Alternative to Magma, Maple, Mathematica and Matlab*, http://wstein.org/papers/icms/icms_2010.pdf, http://www.sagemath.org/.
5. Törbjorn Granlund, *GNU MP Bignum Library*, http://gmplib.org/.
6. William Hart, David Harvey, et al., *Fast Library for Number Theory*, http://www.flintlib.org/.
7. William Hart, Andy Novocin, *A practical univariate polynomial composition algorithm*, (preprint), (2010).
8. William Hart, Andy Novocin, Mark van Hoeij, *Improved polynomial factorisation*, (preprint), (2010).
9. William Hart, Gonzalo Tornaria, Mark Watkins, *Congruent number theta coefficients to* 10^{12}, Proceedings of the Algorithmic Number Theory Symposium (ANTS IX), (Gaudry et. al. eds.) Springer, (to appear), (2010).
10. William Hart, *A One Line Factoring Algorithm*, (preprint), 2010.
11. William Hart, *A refinement of Mulders' polynomial short division algorithm*, (unpublished report), 2007.
12. David Harvey, *A cache–friendly truncated FFT*, Theor. Comput. Sci. 410 (2009), pp. 2649-2658.
13. David Harvey, *Faster polynomial multiplication via multipoint Kronecker substitution*, J. Symb. Comp. 44 (2009), pp. 1502-1510. http://www.cims.nyu.edu/~harvey/code/zn_poly/
14. Thom Mulders, *On short multiplication and division*, AAECC, 11(1), (2000), pp. 69-88.
15. Phong Nguyen, Damien Stehlé, *An LLL algorithm with quadratic complexity*, SIAM Journal of Computation, 39, no. 3, (2009), pp. 874-903. http://perso.ens-lyon.fr/damien.stehle/#software
16. Victor Shoup, *NTL: A Library for doing Number Theory*, http://www.shoup.net/ntl/.